



(12)

EUROPEAN PATENT APPLICATION

- (43)

Date of publication:
29.01.2025 Bulletin 2025/05
- (21)

Application number: 24190414.3
- (22)

Date of filing: 23.07.2024
- (51)

International Patent Classification (IPC):
H04L 69/166 (2022.01) G06F 9/455 (2018.01)
H04L 47/36 (2022.01)
- (52)

Cooperative Patent Classification (CPC):
H04L 69/166; G06F 9/45558; H04L 47/36;
G06F 2009/45595

- (84)

Designated Contracting States:
AL AT BE BG CH CY CZ DE DK EE ES FI FR GB
GR HR HU IE IS IT LI LT LU LV MC ME MK MT NL
NO PL PT RO RS SE SI SK SM TR
Designated Extension States:
BA
Designated Validation States:
GE KH MA MD TN
- (30)

Priority: 24.07.2023 US 202318225638
- (71)

Applicant: VMware LLC
Palo Alto, CA 94304 (US)
- (72)

Inventors:
 - Mathew, Subin Cyriac
Palo Alto, 94304 (US)
- Ang, Boon Seong
Palo Alto, 94304 (US)
 - Jiang, Wenyi
Palo Alto, 94304 (US)
 - Raman, Chidambareswaran
Palo Alto, 94304 (US)
 - Catrouillet, Jerome
Palo Alto, 94304 (US)
 - Song, Sichen
Palo Alto, 94304 (US)
- (74)

Representative: Dilg, Haeusler, Schindelmann
Patentanwalts-gesellschaft mbH
Leonrodstraße 58
80636 München (DE)

(54)

UNDERLAY-BASED TCP SEGMENTATION OFFLOAD IN OVERLAY NETWORKS

(57) Examples described herein include efficient data packet transmission between virtual machines (312, 342, 362), "VMs", on different hosts. An example method includes generating a large data packet at a source VM and determining a modified maximum segment size for efficient transmission. This modified size replaces the default maximum segment size through a TSO MSS override. Segmentation occurs based on the modified size, and the data segments (290) are transmitted to the destination VM (342, 362), even if on a different host. Dynamic determination of the modified size optimizes data transmission efficiency and network performance. It accounts for network headers and enables efficient transmission with or without large receiving offload, "LRO" support. Additionally, non-transitory computer-readable media and servers implementing the method are disclosed. These systems and methods achieve streamlined data transmission, improving network performance and reducing processing overhead.

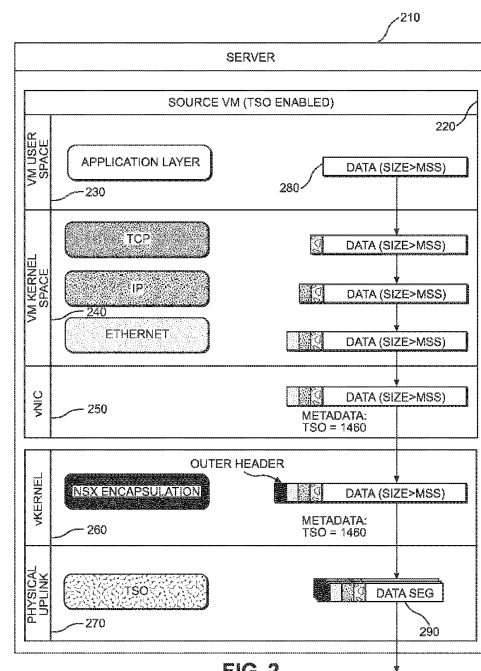


FIG. 2

Description

BACKGROUND

[0001] Modern computer networks attempt to optimize data transmission efficiency and reduce strain on the system's central processing units ("CPUs") whenever possible. To achieve these goals, network interface cards ("NICs") often employ a feature called Transmission Control Protocol ("TCP") segmentation offload ("TSO"). TSO assists in the efficient packaging and transmission of data packets by offloading the segmentation process from the CPU to the NIC hardware.

[0002] TSO is a widely supported feature in modern NICs that helps optimize data transmission in computer networks. By offloading the task of dividing and organizing data packets from the CPU to the NIC hardware, TSO reduces the CPU workload and improves network performance.

[0003] However, in certain network environments such as overlay networks, TSO can face limitations. Overlay networks separate the physical network infrastructure from the virtual workloads running on top of it. In such cases, when the physical, or underlay, network has a much larger capacity for data transmission (known as "maximum transmission unit" or "MTU") compared to the overlay network, traditional TSO becomes a bottleneck.

[0004] The issue arises because traditional TSO uses a smaller value for the TCP maximum segment size ("MSS") based on the limitations of the overlay network. This results in the generation of a large number of small packets when TSO is performed. Consequently, various components in the network, including physical switches, receiver NICs, and network functions at the receiver's virtualization layer, experience increased processing requirements.

[0005] As a result, a need exists for systems and methods that modifies the traditional TSO process based on the capabilities of the underlying physical network. In particular, a need exists for systems and methods for an improved TSO method that dynamically adjusts the TCP maximum segment size based on the underlying physical network, and that maintains transparency of the physical network to the virtual machines ("VMs"), ensuring seamless communication between the VMs and the underlying network infrastructure.

SUMMARY

[0006] Examples described herein include systems and methods for efficiently transmitting data packets between virtual machines on different hosts. An example method includes generating a large data packet at a source VM. The method can include determining a modified maximum segment size for the data packet, which can be based on various factors such as the underlying physical network's maximum segment size or a negotiated end-to-end maximum segment size. The modified

maximum segment size can replace the default maximum segment size for the data packet, which can be replaced by way of a TSO MSS override. The large data packet can be segmented into data segments based on the modified maximum segment size. These data segments can be transmitted to the destination VM, which can be executing on a different physical host relative to the source VM.

[0007] In an example implementation, the determination of the modified maximum segment size and its replacement of the default maximum segment size can be performed dynamically before handling any new large data packet at the source VM. By proactively adjusting the maximum segment size, the method can optimize data transmission efficiency and network performance on the fly, adjusting to the underlying physical network as needed. These adjustments can be made dynamically, such that changes to the underlying physical network are accounted for before additional packets are sent.

[0008] In an example embodiment, the modified maximum segment size can account for the headers added by both the underlying network and the overlay network, ensuring accurate segmentation and efficient transmission. Moreover, if the destination VM supports large receiving offload ("LRO"), the transmitted data segments can be processed at the destination VM without the need for resegmentation. Alternatively, in an example where the destination VM does not support LRO, it may require resegmentation at the destination VM. Even in that example, however, throughput would increase by way of fewer packets being sent across the network.

[0009] This disclosure also includes example non-transitory, computer-readable media containing instructions that, when executed by a hardware-based processor, perform some or all of the method stages described above and elsewhere herein. In another example, a server is disclosed. The server can include a memory storage including a non-transitory, computer-readable medium comprising instructions, and a hardware-based processor that executes the instructions to carry out stages. The processor can carry out some or all of the method stages described above and elsewhere herein, for example. In some examples, multiple servers are utilized to perform various different stages of the method, as described in conjunction with the drawings, below.

[0010] By employing the example systems and methods herein, efficient and streamlined data transmission between VMs on different hosts can be achieved. This can result in improved network performance, reduced processing overhead, and enhanced overall efficiency of data packet delivery.

[0011] Both the foregoing general description and the following detailed description are exemplary and explanatory only and are not restrictive of the examples, as claimed.

BRIEF DESCRIPTION OF THE DRAWINGS

[0012]

FIG. 1 is a flowchart of an example method for efficiently transmitting data packets between VMs on different hosts.

FIG. 2 is a flowchart of an example server sending a data packet using TSO.

FIG. 3 is a flowchart of an example method for efficiently transmitting data packets between VMs on different hosts.

FIG. 4 is a graph showing throughput based on link capacity using the methods described herein versus without using those methods, for a VM that is LRO enabled.

FIG. 5 is a graph showing throughput based on link capacity using the methods described herein versus without using those methods, for a VM that is not LRO enabled.

DESCRIPTION OF THE EXAMPLES

[0013] Reference will now be made in detail to the present examples, including examples illustrated in the accompanying drawings. Wherever possible, the same reference numbers will be used throughout the drawings to refer to the same or like parts.

[0014] Examples described herein include efficient data packet transmission between virtual machines ("VMs") on different hosts. An example method includes generating a large data packet at a source VM and determining a modified maximum segment size for efficient transmission. This modified size replaces the default maximum segment size through a TSO MSS override. Segmentation occurs based on the modified size, and the data segments are transmitted to the destination VM, even if on a different host. Dynamic determination of the modified size optimizes data transmission efficiency and network performance. It accounts for network headers and enables efficient transmission with or without large receiving offload (LRO) support. Additionally, non-transitory computer-readable media and servers implementing the method are disclosed. These systems and methods achieve streamlined data transmission, improving network performance and reducing processing overhead.

[0015] Modern network interface cards offer a rich set of offload features that reduce the CPU's computational load. TSO and LRO significantly reduce the number of packets processed by the OS networking protocol stack to improve TCP performance. With TSO, the operating system ("OS") networking stack can pass a large TCP send buffer directly to the physical NIC, which splits the buffer into smaller packets based on the MSS of that TCP session. On the other hand, LRO enables NICs to aggregate multiple TCP segments into a single large one before passing them to the OS networking stack.

[0016] In the networking environment, traffic from the client VMs can be encapsulated and sent through a physical underlay, which the environment abstracts away from the user to simplify network operations. The users only need to configure the virtual network, or overlay, and the networking environment translates the overlay configuration into a set of underlay data forwarding rules.

[0017] TSO can be performed based on the MSS metadata from the VM networking stack, which is not underlay-aware. It is common to have an overlay MTU much smaller than the underlay MTU. As MTU limits the packet size, overlay MTU affects the MSS of an overlay TCP flow. TSO can therefore produce a large number of TCP packets whose size is much smaller than the underlay MTU. Processing a large number of small packets at the receiver can be a burden for its physical NIC and CPU.

[0018] FIG. 1 provides a flowchart of an example method for efficiently transmitting data packets between VMs on different hosts. Stage 110 can include generating a large data packet at a source VM. In the described method, stage 110 involves the handling of a data packet at the source VM that requires transmission to a destination entity, such as another VM within the virtualized environment. While the term "generation" is used in the context of stage 110, it encompasses the action of preparing a received data packet for subsequent transmission rather than implying its creation from scratch at that VM.

[0019] During this stage, the source VM generates or receives a large data packet, which can originate from various sources within the virtualized environment or even external systems. This data packet may contain a substantial amount of information or a sizable payload that exceeds the size thresholds typically supported by TCP Segmentation Offload (TSO) methods. More specifically, the data packet may be larger than TCP maximum segment size ("MSS") based on the limitations of the overlay network, even though the physical underlay network might be able to handle the packet size.

[0020] In some traditional scenarios, TSO can be performed based on the MSS metadata from the VM networking stack, which traditionally is not underlay-aware. It is common to have an overlay maximum transmission unit (MTU) much smaller than the underlay's MTU. As MTU limits the packet size, overlay MTU affects the MSS of an overlay TCP flow. TSO can therefore produce a large number of TCP packets whose size is much smaller than the underlay MTU. Processing a large number of small packets at the receiver can be a burden for its physical NIC and CPU.

[0021] In stage 120 of the disclosed method, a modified MSS for the data packet can be determined based on the path MTU of the underlying physical system. This step involves determining the modified MSS to optimize transmission efficiency within the network infrastructure. For example, the underlying physical system, comprising network components and devices, sets an MTU that specifies the maximum size of individual packets trans-

mitted over the network. Taking this into consideration, the method dynamically determines an appropriate modified MSS for the data packet independent of the overlay limitations, ensuring efficient transmission without fragmentation or data loss.

[0022] The determination of the modified MSS accounts for the constraints imposed by the path's MTU. Various techniques and algorithms can be employed in the determination process to determine the modified MSS based on the MTU of the data-routing path of the underlying physical system. One example approach involves probing the network to obtain the MTU, which entails sending test packets with different sizes and analyzing the responses received. By measuring the maximum packet size that can be transmitted without fragmentation or errors, the method can determine the appropriate modified MSS for the data packet.

[0023] Additionally, the determination process may utilize preconfigured MTU values specific to the network environment. These values can be predetermined based on the characteristics of the network infrastructure or obtained from network configuration settings. By leveraging these preconfigured MTU values, the method can expedite the determination of the modified MSS, saving time and resources in the process. By determining the MSS based on the path's MTU, the method optimizes transmission efficiency without being constrained by default limits originating from within the virtualized environment.

[0024] Stage 130 of the method can include replacing the default MSS for the data packet with the modified MSS determined at stage 120. Stage 130 can be implemented at the hypervisor kernel level, such that the virtualized components running on top of the hypervisor remain unaware of any change in MSS. To be clear, references to the "hypervisor" with respect to this stage are intended to refer to the hypervisor that controls the execution and resource allocation of the source VM. That is, stage 130 can be performed by the hypervisor rather than being performed by a gateway or other external component for handling network-traffic communications.

[0025] In addition, although the term "replacing" is used to describe adding a modified MSS, this term is explicitly not intended to necessarily include displacing, deleting, or otherwise removing the original MSS. To the contrary, "replacing" can mean adding the modified MSS to the packet header while leaving the original MSS in place or moving the original MSS to another location in the data packet header—in this example, the modified MSS is replacing the original MSS as the operable MSS for data packet transmission while the original MSS remains in an encapsulation header field of the packet header. While this disclosure mentions of adding or modifying a MSS value to or in "the data packet," it should be understood that such a description includes adding or modifying values within metadata associated with the data packet, such as an encapsulation header, rather than altering the payload of the data packet itself.

[0026] By default, the MSS represents the maximum size of a TCP segment within the standard TCP/IP protocol based on the limitations of the overlay network. However, to accommodate the specific requirements and optimizations identified during the previous stages, the method replaces the default MSS with a modified value. The modified MSS reflects the optimized segment size that aligns with the determined path's maximum MTU, but can also include other network characteristics as well. This replacement allows for more efficient transmission and reduces the likelihood of packet fragmentation or unnecessary overhead.

[0027] At the hypervisor kernel level, the necessary modifications are made to the network stack or relevant components to ensure that the data packets transmitted by the VMs adhere to the newly set modified MSS. This can include, for example, inserting information into a packet header indicating the modified MSS value. In another example, this stage includes supplying additional metadata that indicates the modified MSS value.

[0028] At stage 140, the large data packet can be segmented into smaller data segments based on the modified MSS as previously provided by the hypervisor. Each segment is sized according to the modified MSS established in the previous stages of the method. This segmentation process occurs at a physical uplink component, or components, of the hypervisor and is designed to facilitate efficient transmission of the data within the network by minimizing the number of packets transmitted across the physical network.

[0029] For example, if the modified MSS is set to a specific value, the large data packet will be divided into segments of equal or smaller sizes to ensure compliance with the modified MSS. These segments are tailored to fit within the network's maximum MTU and prevent fragmentation or loss of data during transmission.

[0030] The segmentation process can take place at one or more physical uplink components of the hypervisor or network infrastructure, which typically include NICs or other network devices responsible for transmitting data between hypervisors residing on different physical servers. These components are equipped with the necessary capabilities to segment the large data packet into smaller data segments based on the modified MSS.

[0031] At stage 150 of the example method, the segmented data segments are transmitted from the source host to a destination host that executes the destination VM. The destination host can be a different physical server within the networking environment. This stage of the method involves the transfer of the segmented data over the network to ensure reliable delivery and effective communication between VMs.

[0032] The transmission process can follow established networking protocols and mechanisms, such as TCP, for segmenting the original data packet and transmitting the data segments. These protocols ensure the reliable and orderly delivery of the segmented data segments from the source to the destination VM. During

transmission, the network infrastructure routes the segmented data segments through various network components, such as switches, routers, and other networking devices. The network infrastructure ensures that the data segments reach the intended destination VM on the other host.

[0033] Upon receiving the segmented data segments, the destination VM's physical uplink component(s) reassembles them into the original large data packet. This reassembly process aligns with the modified MSS and ensures that the data is reconstructed accurately. If the destination VM supports LRO, the transmitted data segments can be processed at the destination VM without the need for resegmentation. This can include, for example, restoring the original MSS that is stored in an encapsulation field (or other field) of the header of the data packet. Alternatively, the destination VM does not support LRO, it may require resegmentation at the destination VM. Even in that example, however, overall throughput would increase by way of fewer packets being sent across the network.

[0034] In some example networking systems, TSO is not necessarily performed immediately after the VM networking stack sends a large TCP packet with its virtualized network adapter ("vNIC"). Instead, the vNIC of the VM can simply mark the MSS as packet metadata. The packet is then passed to the vSwitch of the hypervisor. TSO is only performed when a large TCP packet is switched to the uplink of the vSwitch. In this example, the packet should already be encapsulated, and the uplink physical NIC of the hypervisor will segment it into a set of smaller packets before transmitting them on the wire. An example of this infrastructure is shown in FIG. 2.

[0035] In more detail, FIG. 2 depicts a server 210 executing a source VM 220. The source VM 220 includes various components, including a VM user space 230, a VM kernel space 240, and a vNIC 250. The VM user space 230 can represent the portion of the source VM 220 that encompasses the user-level applications, processes, and services. It provides a runtime environment where user applications can execute and interact with the underlying virtualization infrastructure.

[0036] In the VM user space 230, various software programs and applications run, enabling users to utilize specific functionalities or services provided by the source VM 220. In the example of FIG. 2, the application layer of the VM user space 230 has generated a data packet 280 with a size that exceeds the MSS associated with the components of the VM 220. This data packet is passed to the VM kernel space 240.

[0037] The VM kernel space 240 includes the core operating system components and functionalities that drive the operation of the source VM 220. It hosts the VM kernel, which is responsible for managing system resources, scheduling tasks, handling memory management, and facilitating communication between the user space and the underlying hypervisor or virtualization platform. The VM kernel space 240 provides an essential

layer of abstraction and control, enabling the source VM 220 to efficiently utilize the physical resources of the server 210.

[0038] As shown, the VM kernel space 240 includes TCP, IP, and Ethernet elements. TCP is responsible for establishing reliable, connection-oriented communication between network endpoints. IP, which stands for Internet Protocol, handles the addressing and routing of data packets across the network. Ethernet, a widely used networking technology, provides the physical transmission and reception of data packets within the network infrastructure. Each of these components can optionally append or modify metadata associated with the data packet 280, as illustrated by the shaded areas added to the lefthand-side of the data packet 280.

[0039] The vNIC 250 of FIG. 2 enables network communication between the source VM 220 and other components in the network. It functions as a virtual representation of a physical NIC, allowing the source VM 220 to send and receive data packets over the network. The vNIC 250 acts as the intermediary between the VM kernel space 240 and the underlying physical network infrastructure, providing network connectivity to the source VM 220. It encapsulates network-related operations and protocols, allowing the source VM 220 to seamlessly exchange data with other VMs, physical hosts, or external networks.

[0040] FIG. 2 also shows components related to a hypervisor executing on the server 210, including a kernel layer 260 and a physical uplink layer 270. The kernel can be located within the hypervisor and is responsible for managing and overseeing various virtualization functions. In the context of network operations, the kernel layer 260 performs virtual network encapsulation of the data packet 280. This process can include adding an outer header to the data packet 280, encapsulating it within additional metadata that enables virtualized network functionality.

[0041] The physical uplink layer 270 can perform TSO, segmenting the data packet 280 into smaller segments 290 of data that fall within an allowable MSS range. The physical uplink layer 270 can be situated at the network interface level. It can segment the data packet 280 into data segments 290 that fall within an allowable MSS range as indicated in the metadata appended to the data packet 280. In this particular example, the MSS for TSO is 1460 bytes, based on the limitations of the virtualized overlay rather than the physical underlay.

[0042] After the physical uplink layer 270 performs TSO and creates appropriately sized data segments 290, it can transmit those data segments 290 across the network to another server housing a destination VM.

[0043] While the example of FIG. 2 involves using a TSO MSS based on the limitations of the virtualized overlay, FIG. 3 provides an example that includes modifying the MSS based on the capabilities of the physical underlay.

[0044] In more detail, FIG. 3 depicts a source server

310 and a destination server 340. The source server 310 includes a source VM 312, which generates data to be sent elsewhere. Destination server 340 includes two different destinations VMs 342, 362. One destination VM 342 is LRO enabled, which allows it to process large data packets. Meanwhile, destination VM 362 is LRO disabled, meaning it cannot process the same size data packets, and instead requires further segmentation of data packets before processing. The example of FIG. 3 provides examples of sending data from a source VM 312 to both of the destination VMs 342, 362.

[0045] The various VMs 312, 342, 362 shown in FIG. 3 include various respective components, including a VM user space 314, 344, 364, a VM kernel space 318, 346, 366, and a vNIC 318, 348, 368. The VM user space 314, 344, 364 can represent the portion of the VM 312, 342, 362 that encompasses the user-level applications, processes, and services. It provides a runtime environment where user applications can execute and interact with the underlying virtualization infrastructure.

[0046] In the VM user space 314, 344, 364, various software programs and applications run, enabling users to utilize specific functionalities or services provided by the respective VM 312, 342, 362. In the example of FIG. 3, the application layer of the source VM's 312 user space 314 has generated a data packet 380 with a size that exceeds the MSS associated with the components of the VM 312. This data packet is passed to the VM kernel space 316.

[0047] The VM kernel space 316, 346, 366 includes the core operating system components and functionalities that drive the operation of the respective VMs 312, 342, 362. It hosts the VM kernel, which is responsible for managing system resources, scheduling tasks, handling memory management, and facilitating communication between the user space and the underlying hypervisor or virtualization platform. The VM kernel space 316, 346, 366 provides an essential layer of abstraction and control, enabling the VMs 312, 342, 362 to efficiently utilize the physical resources of the servers 310, 340.

[0048] As shown, the VM kernel space 316, 346, 366 includes TCP, IP, and Ethernet elements. TCP is responsible for establishing reliable, connection-oriented communication between network endpoints. IP, on the other hand, handles the addressing and routing of data packets across the network. Ethernet, a widely used networking technology, provides the physical transmission and reception of data packets within the network infrastructure. Each of these components can optionally append or modify metadata associated with the data packet 380 after it is created by the application layer at stage 381. The shaded areas added to the lefthand-side of the data packet 280 are intended to represent appended or modified metadata of the data packet 380 in this example. Similarly, stages 395 and 398 can include processing and removing the metadata as necessary before providing the data packet (or packets) to the respective application layer.

[0049] The vNIC 318, 348, 368 of each respective VM 312, 342, 362 enables network communication between the VMs 312, 342, 362 and other components in the network. It functions as a virtual representation of a physical NIC, allowing the VMs 312, 342, 362 to send and receive data packets over the network. The vNIC 318, 348, 368 acts as the intermediary between the VM kernel space 316, 346, 366 and the underlying physical network infrastructure, providing network connectivity to the VMs 312, 342, 362. It encapsulates network-related operations and protocols, allowing the VMs 312, 342, 362 to seamlessly exchange data with other VMs, physical hosts, or external networks. These functions can be represented by stage 382 in the example of the source VM 312, and stages 394 and 397 in the example of the destination VMs 342, 362.

[0050] FIG. 3 also shows components related to hypervisors executing on the servers 310, 340, including a kernel layer 320, 350 and a physical uplink layer 322, 352. Each kernel can be located within a respective hypervisor and is responsible for managing and overseeing various virtualization functions. In the context of network operations, the kernel layer 320, 350 performs virtual network encapsulation of the data packet 380. This process can include adding an outer header to the data packet 280, encapsulating it within additional metadata that enables virtualized network functionality. This process can be represented by stage 383, for example.

[0051] In the example of FIG. 3, at stage 383 the kernel layer 320 of the hypervisor overrides the TSO MSS previously set in the metadata at stage 381. More specifically, the TSO MSS in this example was originally set to 1460 bytes, similar to the example of FIG. 2. This MSS limit can be implemented based on the limitations of the virtualized overlay components. However, in this example, the physical underlay components are capable of sending larger data packets. As a result, the kernel layer 320 overrides the TSO MSS and sets a new value-in this example, 8760 bytes. The override step can be performed by modifying or inserting new metadata appended to the data packet 380 to create a modified data packet 390. Although the kernel layer 320 includes an element titled "NSX encapsulation," this element can be any overlay module of the kernel layer and is not specific to any particular platform.

[0052] The override step at stage 383 can be performed by the hypervisor that controls the execution and resource allocation of the source VM, rather than being performed by a gateway or other external component for handling network-traffic communications. In addition, this step can include creating or modifying packet metadata to include the new MSS value. The new MSS value is then passed to the physical uplink layer as packet metadata. The original MSS can be saved in the encapsulation header so that the packet can be resegmented at the receiver side, such as where the receiving VM does not support LRO.

[0053] The modified data packet 390, which can include the new MSS value as packet metadata, can then be passed to the physical uplink layer 322 at stage 384. The physical uplink layer 322 can be the hypervisor software which understands the capabilities of the physical NIC and adjusts the packets based on those capabilities before sending and after receiving the packets to or from the physical NIC. For example, the physical uplink layer 322 is responsible for performing TSO if the physical NIC does not support TSO. If the physical NIC supports TSO, the physical uplink layer will pass the large packet to the physical NIC, but will also inform it of the MSS for that packet, as a buffer descriptor or other form of metadata, so that the physical NIC can perform the TSO.

[0054] In this example, at stage 391, the hypervisor of the source VM 310 transmits the modified data packet 390 to the hypervisor of the destination VM 340. The physical uplink layer 352 of the destination VM 340 can then transfer the data packet 390 to the kernel layer 350. At the kernel layer 350 of the receiving hypervisor, the TSO MSS can be overridden again, this time to return the value to its original value as set by the source VM 312. By returning the TSO MSS value to the original value, the MSS modification remains invisible to the VMs 312, 342, 362 involved in the data transmission process.

[0055] FIG. 3 shows two variations in an example method depending upon whether the receiving VM 342, 362 supports LRO. For example, if the receiving VM 342 supports LRO, then at stage 393 the kernel layer 350 can transmit data packet 390 in its large form; that is, without segmenting the data packet 390. In that example, the vNIC 348 of the destination VM 342 provides the LRO functionality and handles the data packet 390, passing it to the VM kernel space 346 at stage 394. The data packet 390 can then be stripped of any headers or other metadata that are not required for the application layer, and the resulting data packet 390 can then be passed to the application layer at stage 395. At that point, the transmission procedure is complete for the data packet 380 created at the source VM 312.

[0056] In another example, the destination VM 362 does not support LRO. In that example, at stage 396, the kernel space 350 passes the large data packet 390 to the vNIC 368 of the destination VM 362. The vNIC 368 can then segment the data packet 390 into smaller data segments that fall within the TSO MSS of 1460 bytes, as shown. The smaller-sized data segments can then be passed to the VM kernel space 366 at stage 397. These data segments can then be stripped of any headers or other metadata that are not required for the application layer, and the resulting data segments can then be passed to the application layer at stage 398.

[0057] Although FIG. 3 shows only one data packet 390 sent between the source server 310 and destination server 340, the method can include sending multiple data packets 390. For example, the overall package of data being sent between the servers 310, 340 can be 2 MB. With a maximum packet size of 1460 bytes, this overall

data package would require the transmission of 1437 individual packets. But with a maximum packet size of 8760 bytes based on the limits of the physical components involved in the transmission path, the same data packet would require only 240 individual packets. This greatly reduces the load on the physical components in the network and makes data transmission faster and more efficient.

[0058] FIG. 4 shows example results from a real-world test of a system as reflected in FIG. 3, with an LRO-enabled destination VM. As the link capacity increases, the system moves from being "network-bounded" to being "CPU-bounded." As shown in the graphs of FIG. 4, before the link capacity reaches 11 Gb/s, both the baseline and the improved design described herein have the TCP throughput reaching the link capacity. Meanwhile, CPU usage gradually increases as throughput increases. The link capacity limits the TCP throughput. In this case, the improved design shows a large CPU usage reduction while achieving the same throughput. For example, at link capacity being 11Gb/s, both implementations achieve 98% bandwidth utilization. However, the improved design uses 49.6% of a CPU core while the baseline already fully uses CPU.

[0059] Dynamic TSO MSS overriding achieves reduced CPU utilization by sending a smaller number of larger packets to the underlay. When encapsulated packets reach the destination hypervisor, each packet must go through a set of NSX processing steps, including decapsulation, switching, and applying firewall checks. Thus, a reduced number of packets results in a reduced CPU load. The saved CPU resources can be used by the workload VMs to achieve improved performance.

[0060] As the link capacity increases beyond 11Gb/s, the baseline implementation is limited by the CPU and has its TCP throughput remain at 11Gb/s. For the improved design, the throughput increases with the link capacity until 21Gb/s. Afterward, the improved design starts to be bounded by the CPU, and the TCP throughput stays the same.

[0061] When both implementations are CPU bounded, the improved design as described herein achieves higher throughput. The existing implementation can achieve at most 11.1Gb/s while 21.4Gb/s is achieved with dynamic TSO MSS overriding. More efficient use of the CPU contributes to these throughput improvements.

[0062] FIG. 5 shows a similar graph showing results from a similar test, but for a system where LRO is disabled at the destination VM. When the LRO is disabled at the destination VM, each packet passed to its OS networking stack must comply with its MTU, which can be a 1500-byte limit for example. Compared to having LRO enabled, performance decreases due to the increased number of I/O operations. The performance improvement of the improved design over the baseline is also reduced when LRO is disabled, due to the overhead of software-based packet re-segmentation at the vNIC back end. These experiment results when LRO is disabled are shown in

FIG. 5.

[0063] Nevertheless, the experimental results look similar to those of FIG. 4. That is, compared to the baseline, dynamic TSO MSS overriding reduces the CPU load and can achieve higher throughput. The baseline design is CPU bounded at 6.9Gb/s while the improved design reaches 12.7Gb/s. An 84% improvement is achieved. Therefore, disabling LRO reduces the throughput improvement of the improved design from 92% to 84% in the example experiment. The decrease is likely due to software-based re-segmentation only being applied when LRO is disabled. Throughput is still significantly improved in this case, thanks to the reduced number of packets processed by the networking stack at the hypervisor kernel.

[0064] Other examples of the disclosure will be apparent to those skilled in the art from consideration of the specification and practice of the examples disclosed herein. Though some of the described methods have been presented as a series of steps, it should be appreciated that one or more steps can occur simultaneously, in an overlapping fashion, or in a different order. The order of steps presented is only illustrative of the possibilities and those steps can be executed or performed in any suitable fashion. Moreover, the various features of the examples described here are not mutually exclusive. Rather any feature of any example described here can be incorporated into any other suitable example. It is intended that the specification and examples be considered as exemplary only, with a true scope and spirit of the disclosure being indicated by the following claims.

Claims

1. A method for efficiently transmitting data packets (280, 380, 390) between virtual machines (312, 342, 362), "VMs", on different hosts, comprising:

generating a data packet (280, 380, 390) at a source VM (312) on a source host, the data packet (280, 380, 390) including in a header field a value indicating a default maximum segment size, "default MSS";
 receiving the data packet (280, 380, 390) at a hypervisor that controls the execution and resource allocation of the source VM (312);
 determining, by the hypervisor, a modified maximum segment size ("modified MSS") for the data packet (280, 380, 390);
 replacing, by the hypervisor, the default MSS for the data packet (280, 380, 390) with the modified MSS;
 segmenting, at the source host, the data packet (280, 380, 390) into data segments (290) based on the modified MSS; and
 transmitting the data segments (290) to the destination VM (342, 362).

2. The method of claim 1, comprising at least one of the following features:

wherein the modified MSS is determined based on an underlay maximum segment size, the underlay maximum segment size corresponding to a maximum segment size associated with an underlying physical network supporting at least the source VM (312);

wherein the modified MSS is determined based on an end-to-end maximum segment size negotiated by the source VM (312) and destination VM (342, 362).

3. The method of claim 1 or 2, wherein the default MSS is inserted into an other portion of the data packet (280, 380, 390).

4. The method of claim 3, wherein the other portion of the data packet (280, 380, 390) is an encapsulation header field.

5. The method of any of claims 1 to 4, comprising at least one of the following features:

wherein the destination VM (342, 362) is capable of large receiving offload, "LRO", and wherein the method further comprises, at the destination VM (342, 362), processing the transmitted data segments (290) without performing resegmentation on those data segments (290); wherein determining and replacing are performed before handling a new data packet (280, 380, 390) generated at the source VM (312).

6. A non-transitory, computer-readable medium containing instructions that, when executed by a hardware-based processor, cause the processor to perform stages for efficiently transmitting data packets (280, 380, 390) between virtual machines (312, 342, 362), "VMs", on different hosts, the stages comprising:

generating a data packet (280, 380, 390) at a source VM (312) on a source host, the data packet (280, 380, 390) including in a header field a value indicating a default maximum segment size, "default MSS";
 receiving the data packet (280, 380, 390) at a hypervisor that controls the execution and resource allocation of the source VM (312);
 determining, by the hypervisor, a modified maximum segment size, "modified MSS" for the data packet (280, 380, 390);
 replacing, by the hypervisor, the default MSS for the data packet (280, 380, 390) with the modified MSS;

- segmenting, at the source host, the data packet (280, 380, 390) into data segments (290) based on the modified MSS; and
transmitting the data segments (290) to the destination VM (342, 362). 5
7. The non-transitory, computer-readable medium of claim 6, comprising at least one of the following features: 10
- wherein the modified MSS is determined based on an underlay maximum segment size, the underlay maximum segment size corresponding to a maximum segment size associated with an underlying physical network supporting at least the source VM (312); 15
- wherein the modified MSS is determined based on an end-to-end maximum segment size negotiated by the source VM (312) and destination VM (342, 362). 20
8. The non-transitory, computer-readable medium of claim 6 or 7, wherein the default MSS is inserted into an other portion of the data packet (280, 380, 390). 25
9. The non-transitory, computer-readable medium of claim 8, wherein the other portion of the data packet (280, 380, 390) is an encapsulation header field. 30
10. The non-transitory, computer-readable medium of any of claims 6 to 9, comprising at least one of the following features: 35
- wherein the destination VM (342, 362) is capable of large receiving offload, "LRO", and wherein the stages further comprise, at the destination VM (342, 362), processing the transmitted data segments (290) without performing resegmentation on those data segments (290); wherein determining and replacing are performed before handling a new data packet (280, 380, 390) generated at the source VM (312). 40
11. A server for efficiently transmitting data packets (280, 380, 390) between virtual machines (312, 342, 362), "VMs", comprising: 45
- a memory storage including a non-transitory, computer-readable medium comprising instructions; and 50
- a hardware-based processor that executes the instructions to carry out stages comprising: 55
- generating a data packet (280, 380, 390) at a source VM (312) on a source host, the data packet (280, 380, 390) including in a
- header field a value indicating a default maximum segment size ("default MSS");
receiving the data packet (280, 380, 390) at a hypervisor that controls the execution and resource allocation of the source VM (312);
determining, by the hypervisor, a modified maximum segment size ("modified MSS") for the data packet (280, 380, 390);
replacing, by the hypervisor, the default MSS for the data packet (280, 380, 390) with the modified MSS;
segmenting, at the source host, the data packet (280, 380, 390) into data segments (290) based on the modified MSS; and
transmitting the data segments (290) to the destination VM (342, 362).
12. The server of claim 11, comprising at least one of the following features:
- wherein the modified MSS is determined based on an underlay maximum segment size, the underlay maximum segment size corresponding to a maximum segment size associated with an underlying physical network supporting at least the source VM (312);
wherein the modified MSS is determined based on an end-to-end maximum segment size negotiated by the source VM (312) and destination VM (342, 362).
13. The server of claim 11 or 12, wherein the default MSS is inserted into an other portion of the data packet (280, 380, 390).
14. The server of claim 13, wherein the other portion of the data packet (280, 380, 390) is an encapsulation header field.
15. The server of any of claims 11 to 14, wherein determining and replacing are performed before handling a new data packet (280, 380, 390) generated at the source VM (312).

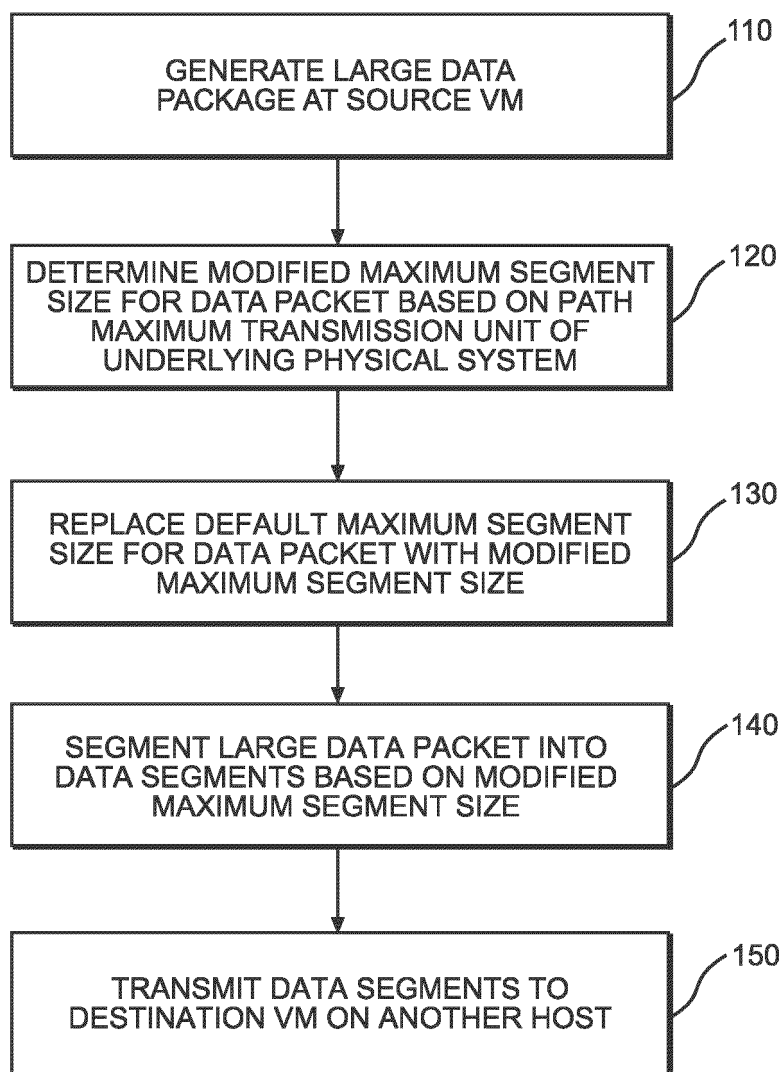


FIG. 1

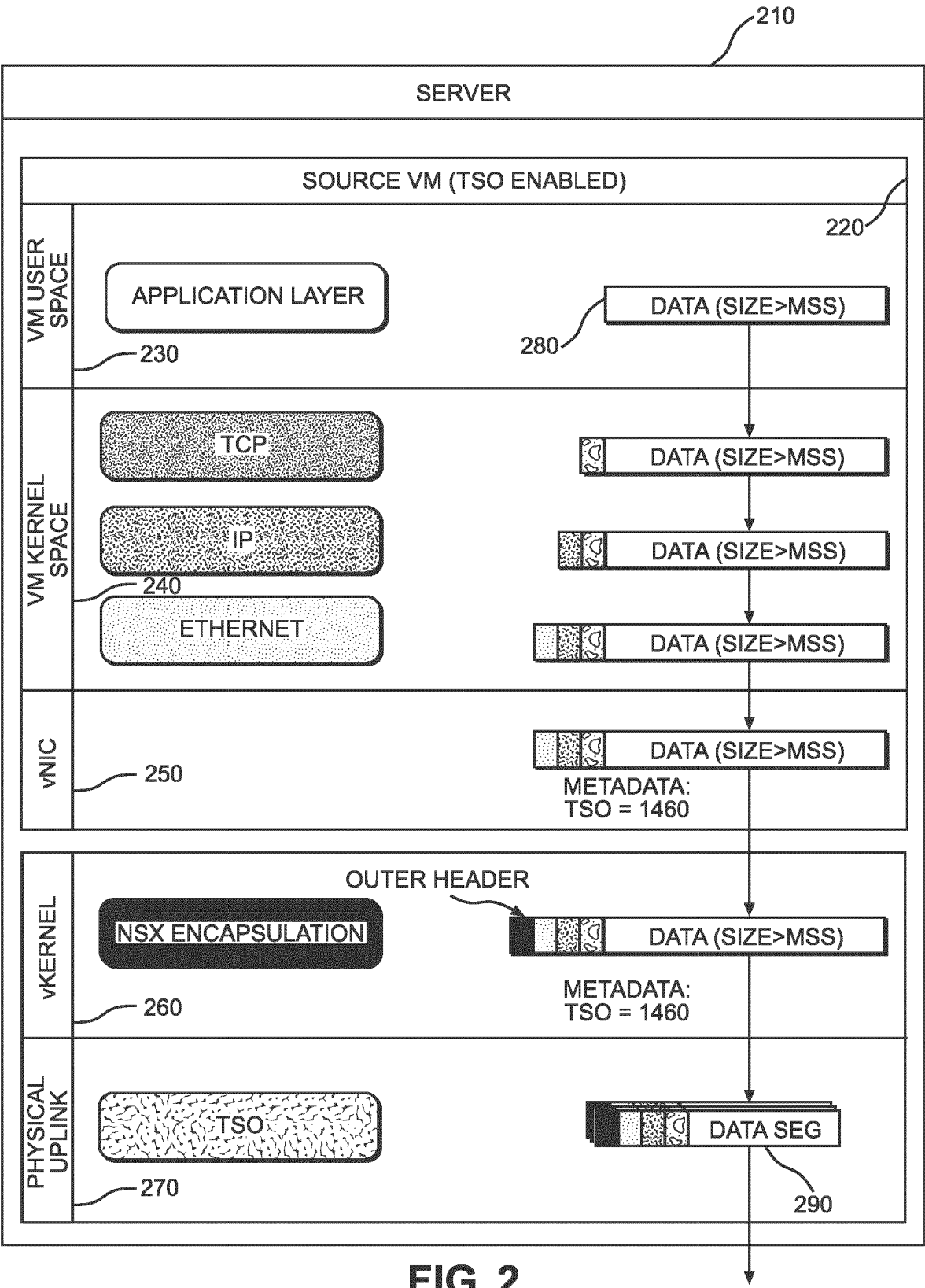


FIG. 2

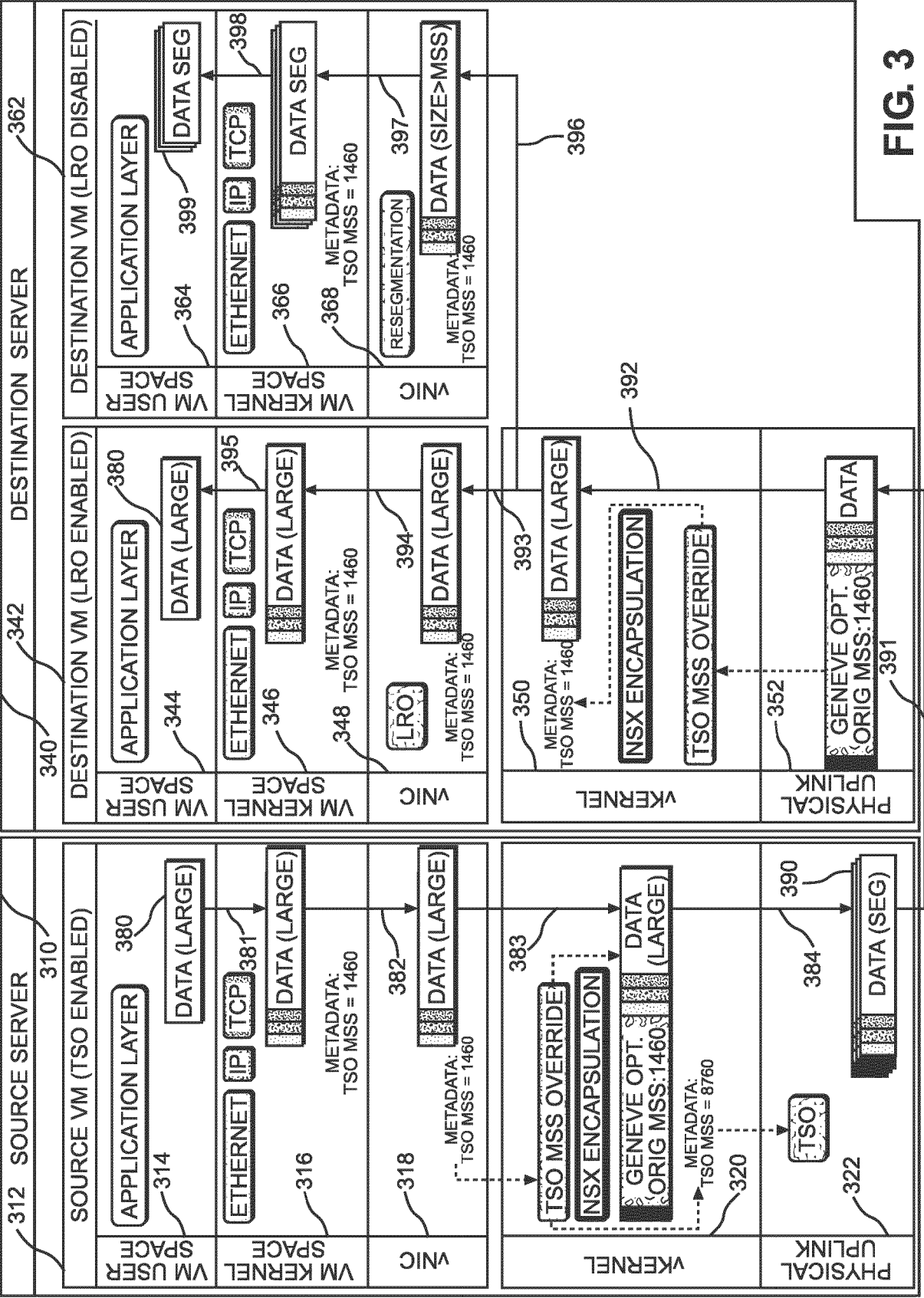
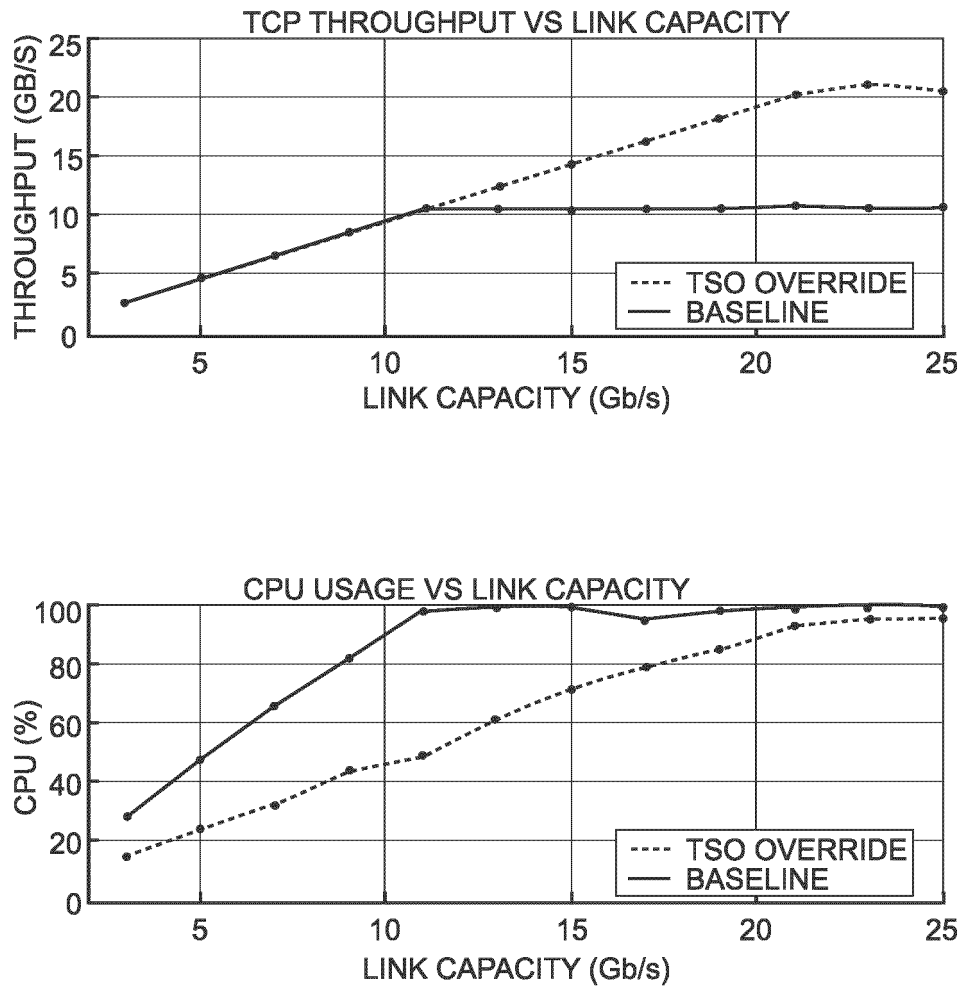


FIG. 3

**FIG. 4**

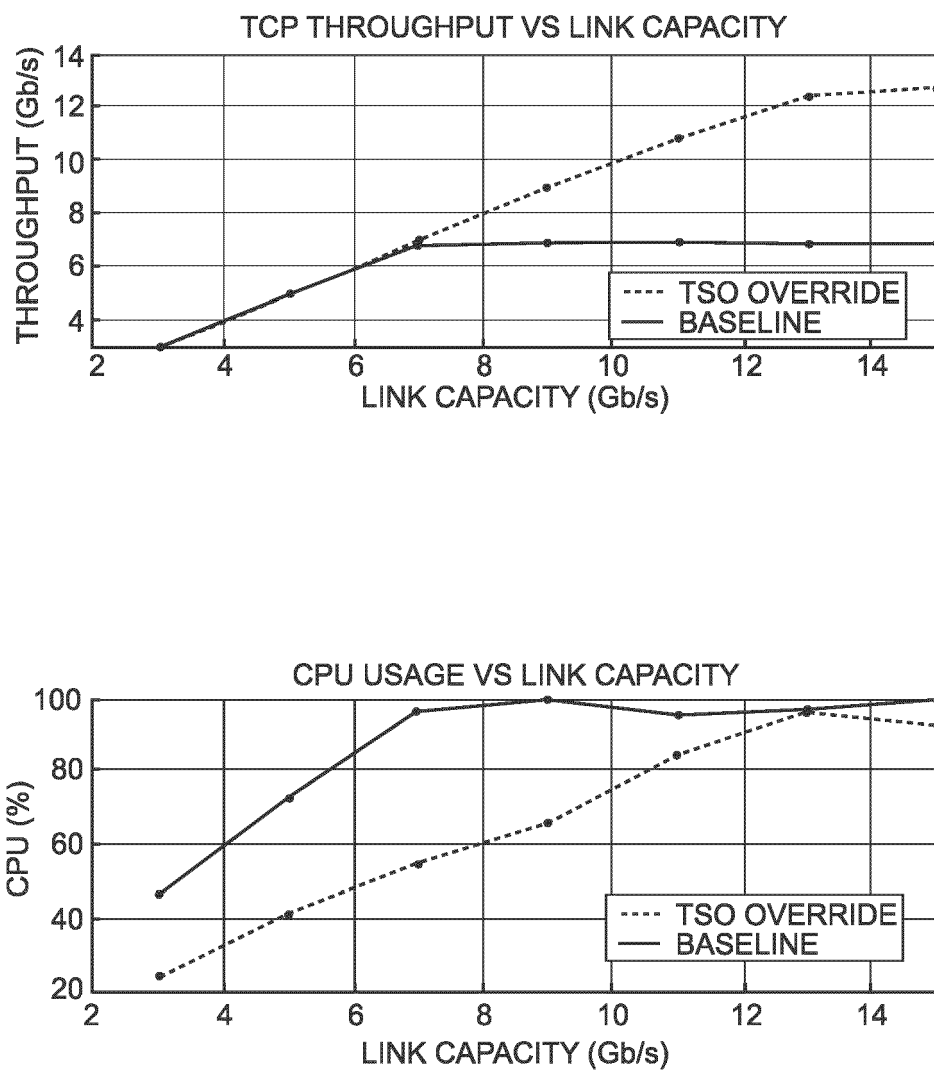


FIG. 5



EUROPEAN SEARCH REPORT

Application Number

EP 24 19 0414

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (IPC)
X	US 2019/132296 A1 (JIANG WENYI [US] ET AL) 2 May 2019 (2019-05-02) * abstract * * figures 1, 2 * * paragraph [0016] - paragraph [0041] * -----	1 - 15	INV. H04L69/166 G06F9/455 H04L47/36
			TECHNICAL FIELDS SEARCHED (IPC)
			H04L G06F
The present search report has been drawn up for all claims			
Place of search		Date of completion of the search	Examiner
The Hague		9 October 2024	Hornik, Valentin
CATEGORY OF CITED DOCUMENTS			
X : particularly relevant if taken alone Y : particularly relevant if combined with another document of the same category A : technological background O : non-written disclosure P : intermediate document			
T : theory or principle underlying the invention E : earlier patent document, but published on, or after the filing date D : document cited in the application L : document cited for other reasons & : member of the same patent family, corresponding document			

EPO FORM 1503 03.82 (P04C01)

This annex lists the patent family members relating to the patent documents cited in the above-mentioned European search report. The members are as contained in the European Patent Office EDP file on

The European Patent Office is in no way liable for these particulars which are merely given for the purpose of information.

09 - 10 - 2024

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2019132296 A1	02-05-2019	US 2019132296 A1	02-05-2019
		US 2022191181 A1	16-06-2022
		US 2023396598 A1	07-12-2023

EPO FORM P0459

For more details about this annex : see Official Journal of the European Patent Office, No. 12/82